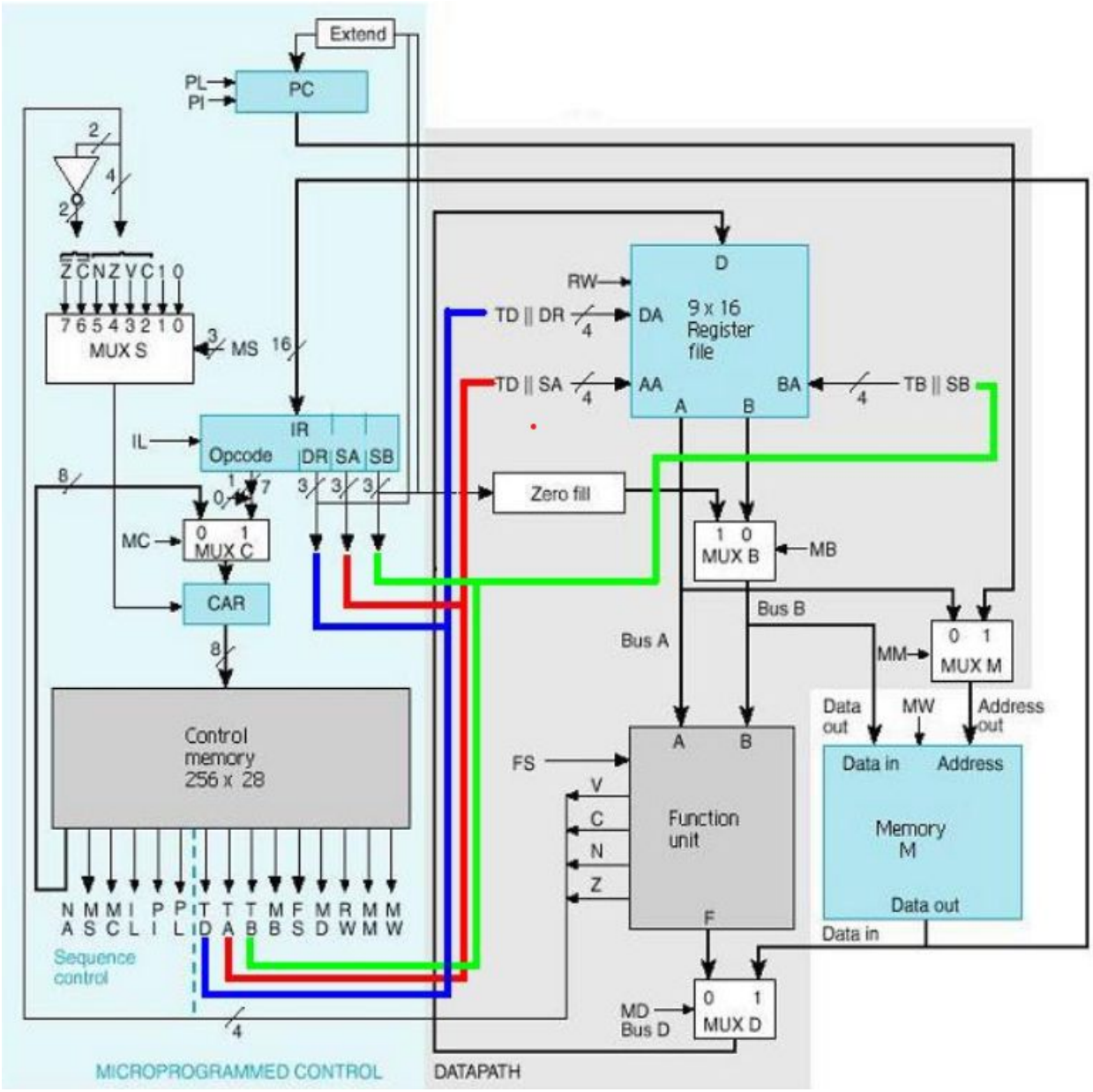
CSU22022: COMPUTER ARCHITECTURE I

MICROCODED INSTRUCTION SET PROCESSOR

LEXES JAN MANTIQUILLA - GitHub



Introduction

In this assignment we were tasked to implement a microprogrammed instruction set processor.

Testbench results

Processor

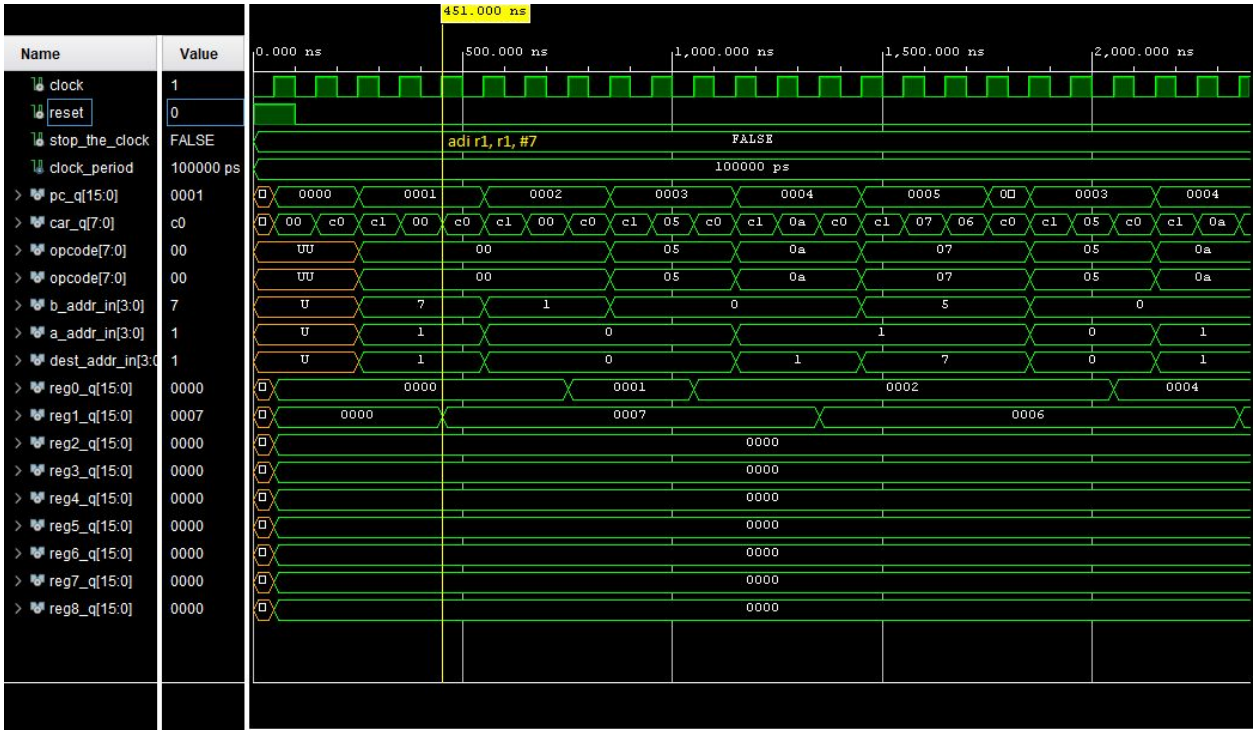
The final processor contains all of the entities and is capable of executing the instructions placed into the memory starting from memory location 0. The instructions implemented are **`ADI, LD, ST, INC, NOT, ADD, B, BNE, NOP, SR, DEC`**.

The memory contains a small sample program, the pseudocode of which is as follows:

```
adi r1, r1, #7 ; i = 7
adi r0, r0, #1 ; num = 1
               ; while (i >= 0) {
add r0, r0, r0 ;   num += num
dec r1, r1     ;   i--
bne -3, r1     ; }
st [r0], r0    ; Mem.halfword[num] = num
ld r1, [r0]    ; num_copy = Mem.halfword[num]
inc r1, r1     ; num_copy++
not r1, r1     ; num_copy ~= num_copy
               ; while (true) {
sr r1, r1      ;   num_copy >>= num_copy
b -2           ; }
```

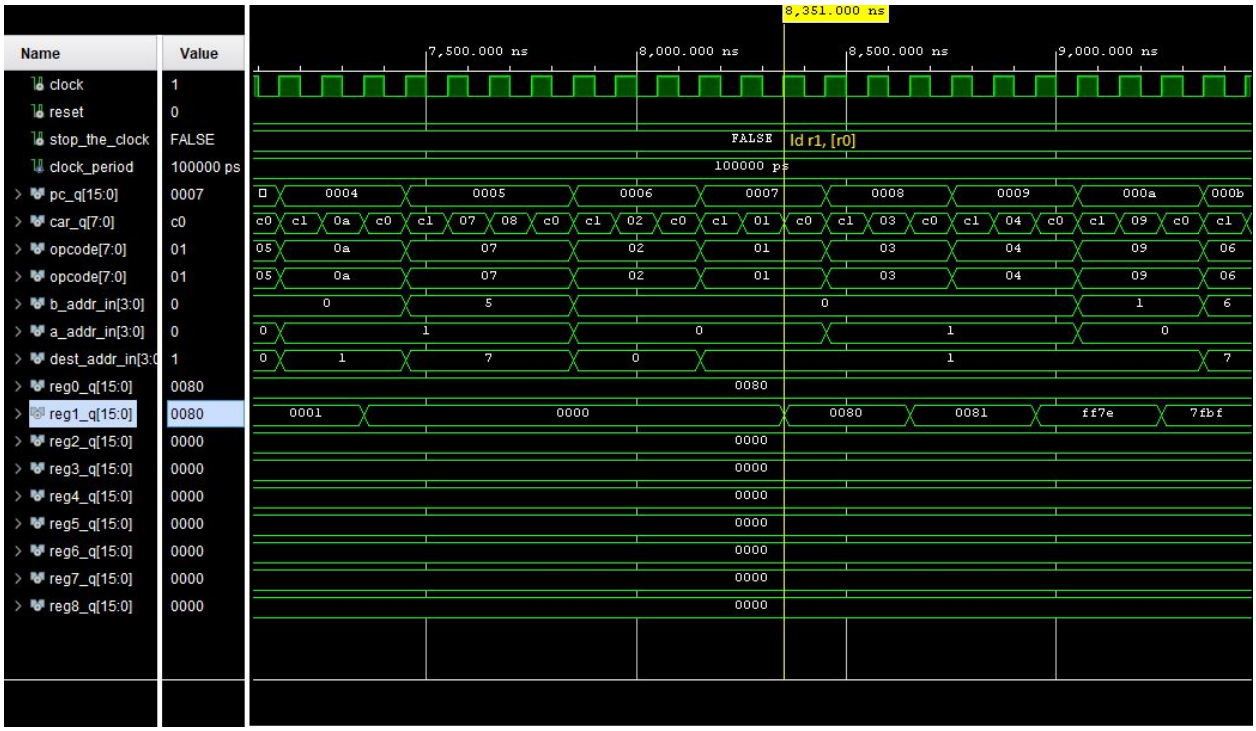
Instructions implemented

ADI



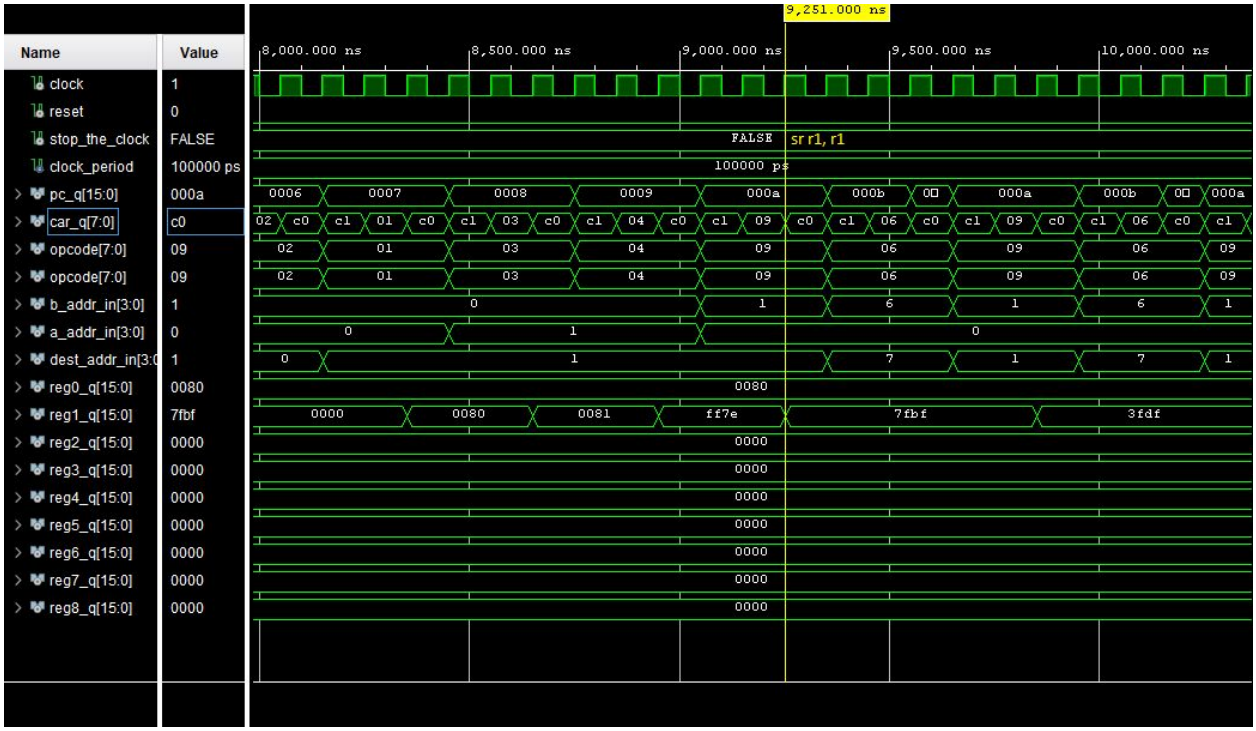
The above image illustrates the add immediate (**`adi`**) instruction implemented. It added the number 0x7 to the register 1

LD



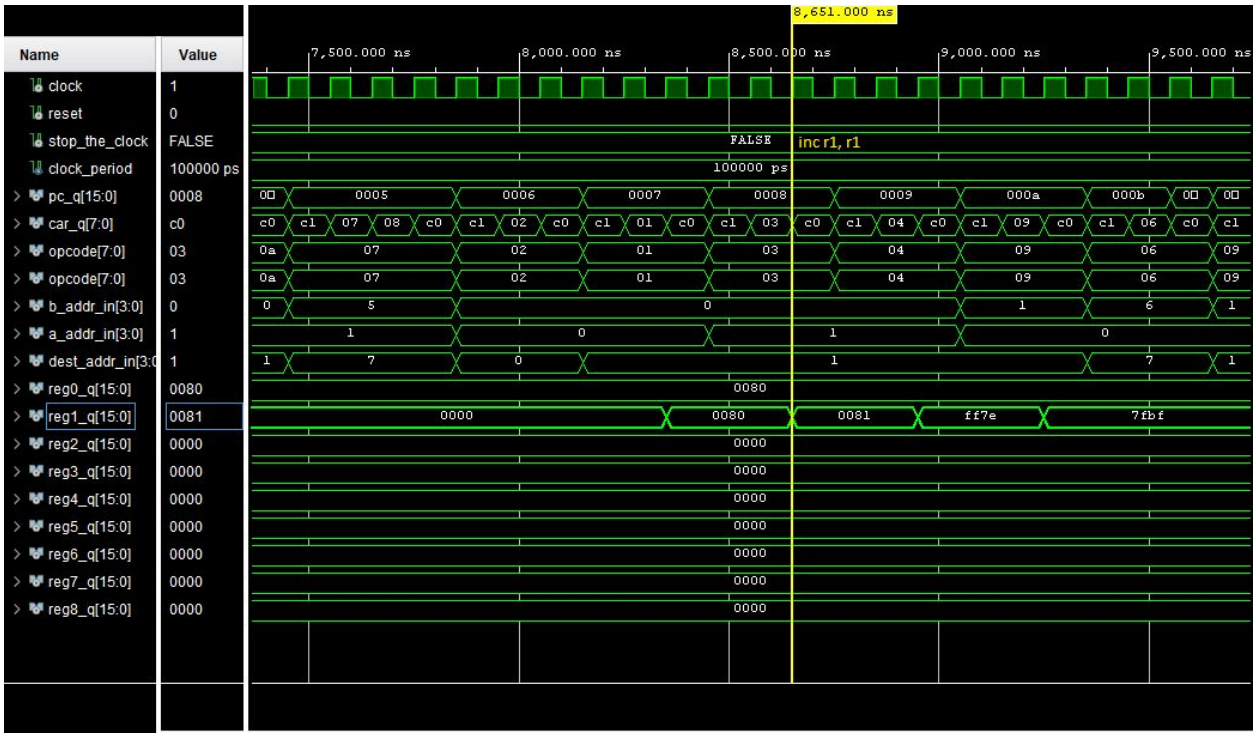
The above image illustrates the load (**`ld`**) instruction implemented. It loaded the number 0x0080 (from memory location 0x0080) into register 1

SR



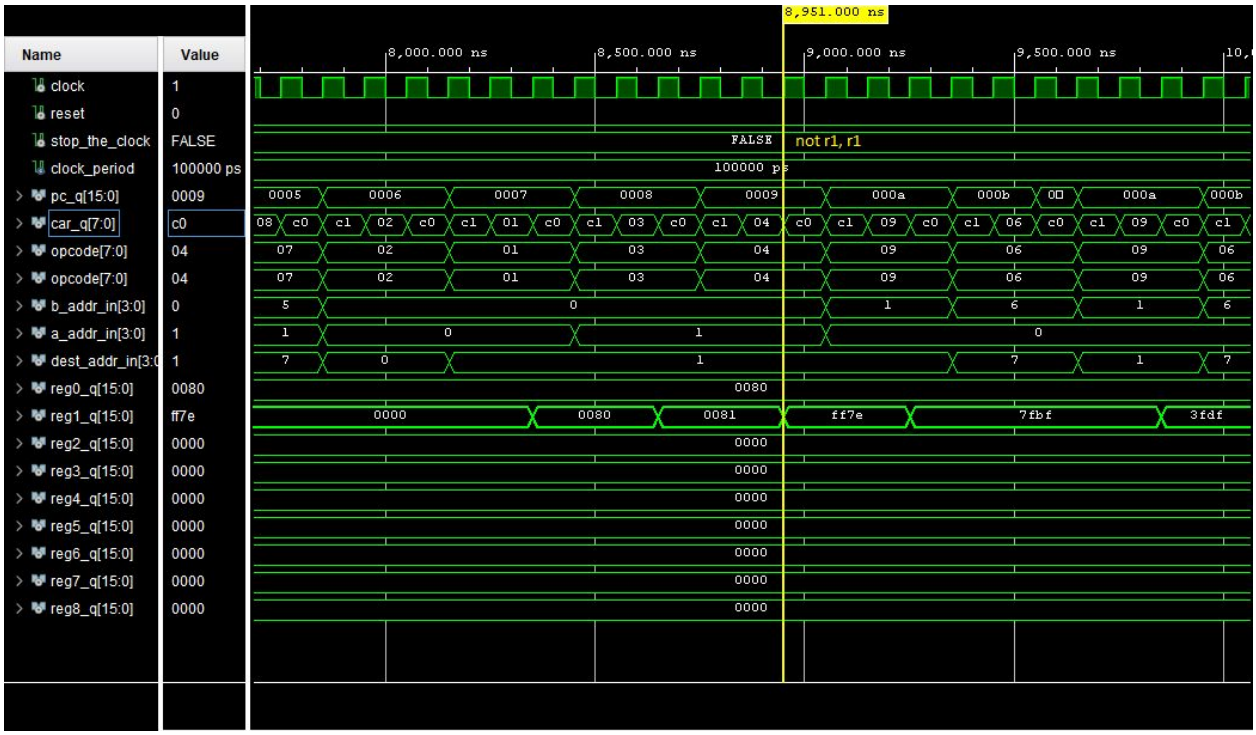
The above image illustrates the shift right (**sr**) instruction implemented. It shifted the number (0xff7e) in register 1 to the right by 1 and stored the result (0x7fbf) into register 1.

INC



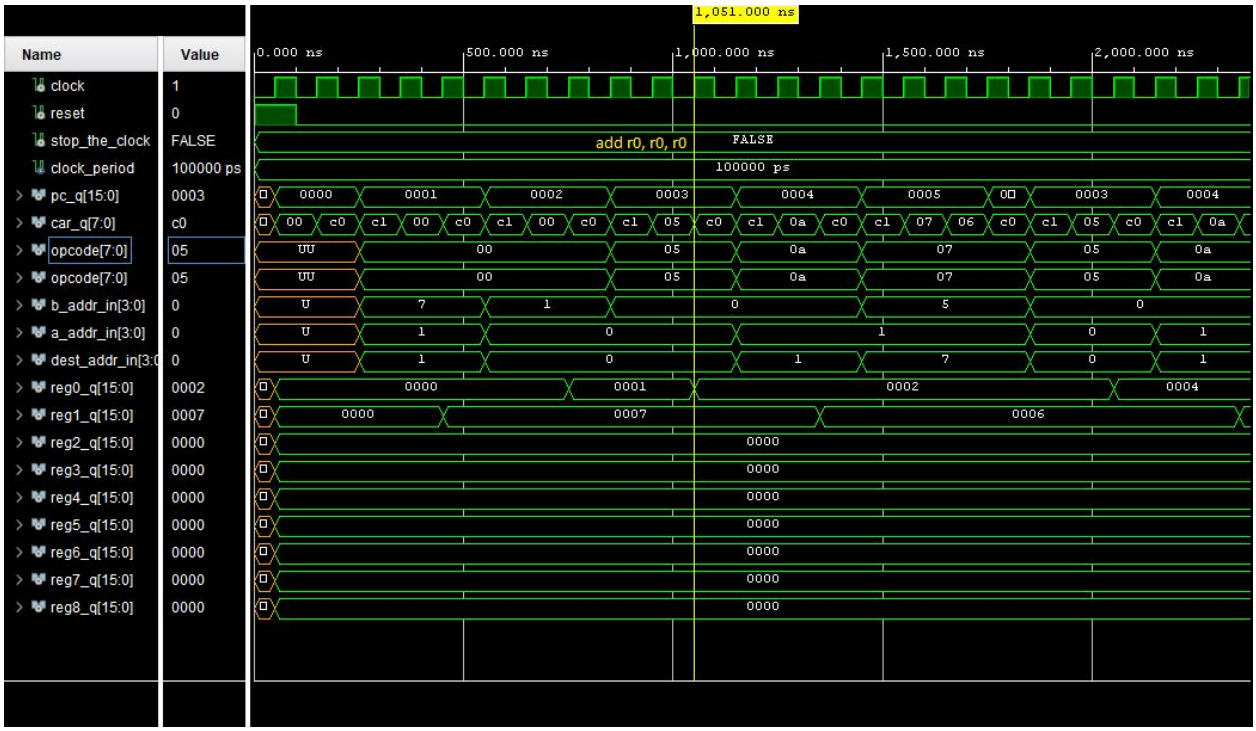
The above image illustrates the increment (**inc**) instruction implemented. It incremented the number 0x0080 to 0x0081.

NOT



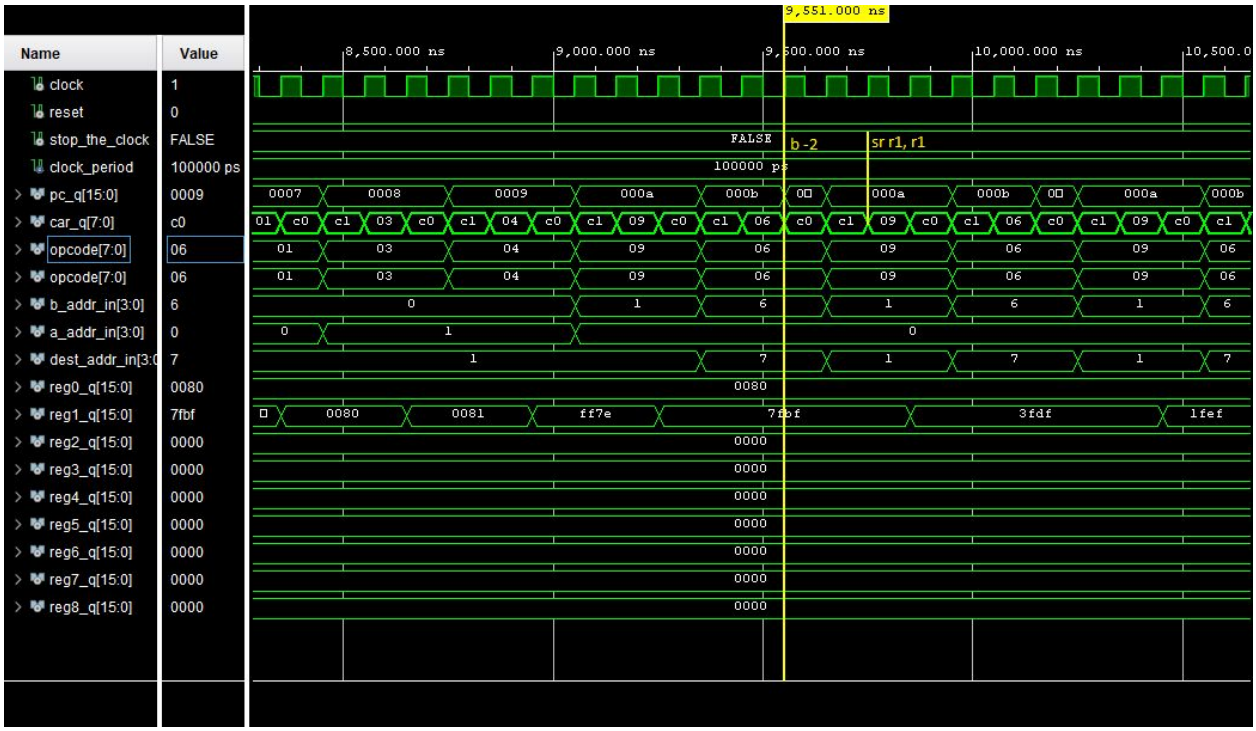
The above image illustrates the **`not`** instruction implemented. It applied the bitwise not to the number (0x0081) in register 1 and stored the result (0xff7e) in register 1.

ADD



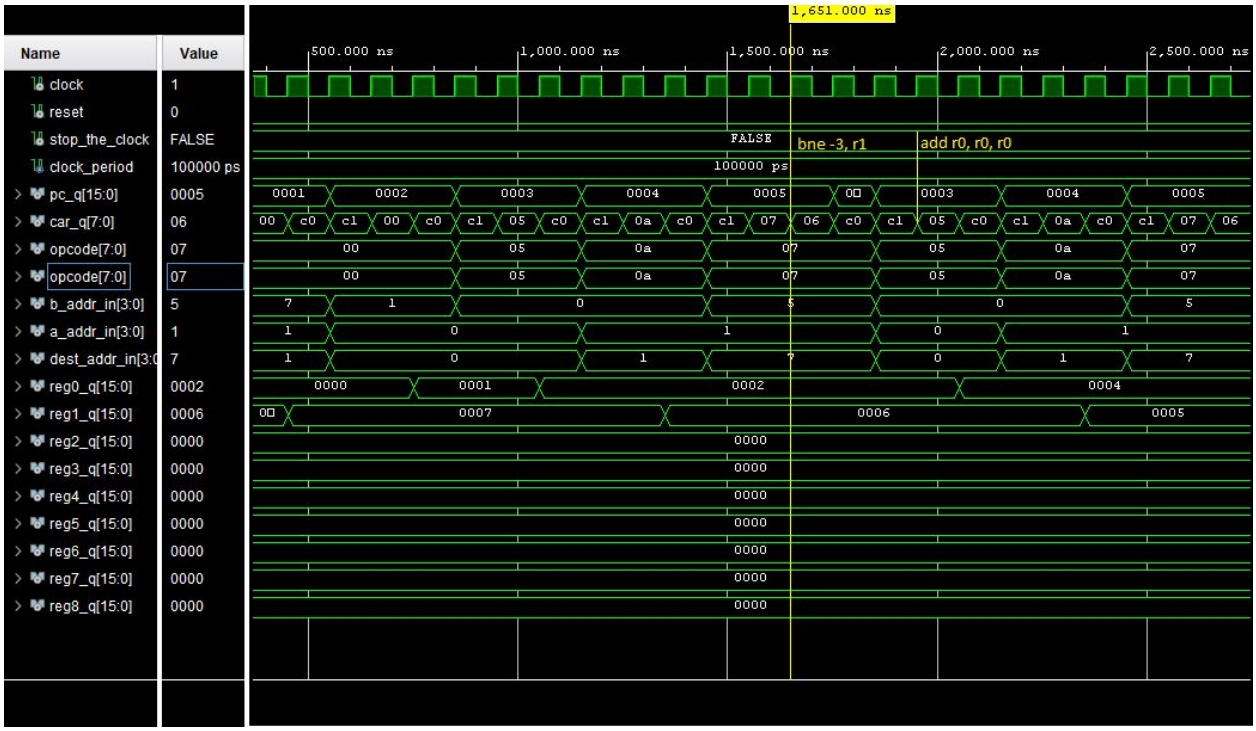
The above image illustrates the **add** instruction implemented. It added the number (0x1) in register 0 with the number (0x1) in register 0 and stored the result (0x2) in register 0.

B (unconditional jump)



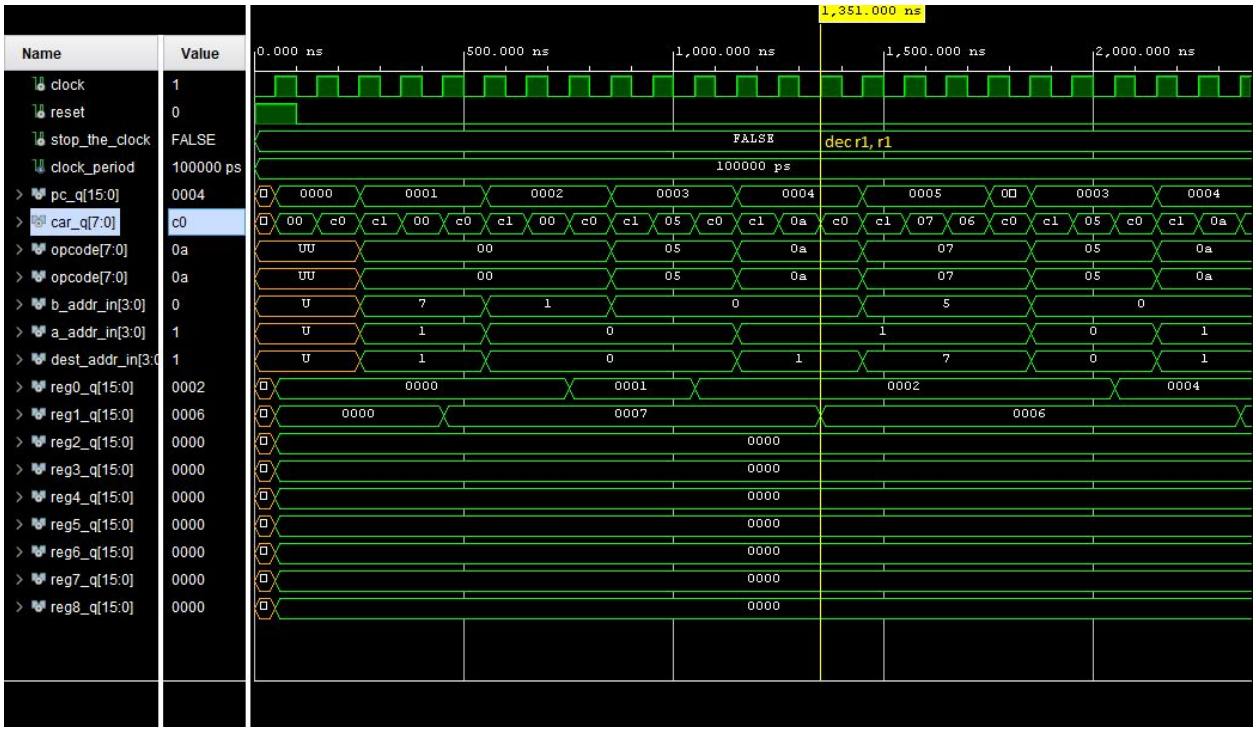
The above image illustrates the **b** or branch (unconditional jump) instruction implemented. It branched back to the previous instruction (**sr** r1, r1).

BNE (conditional jump)



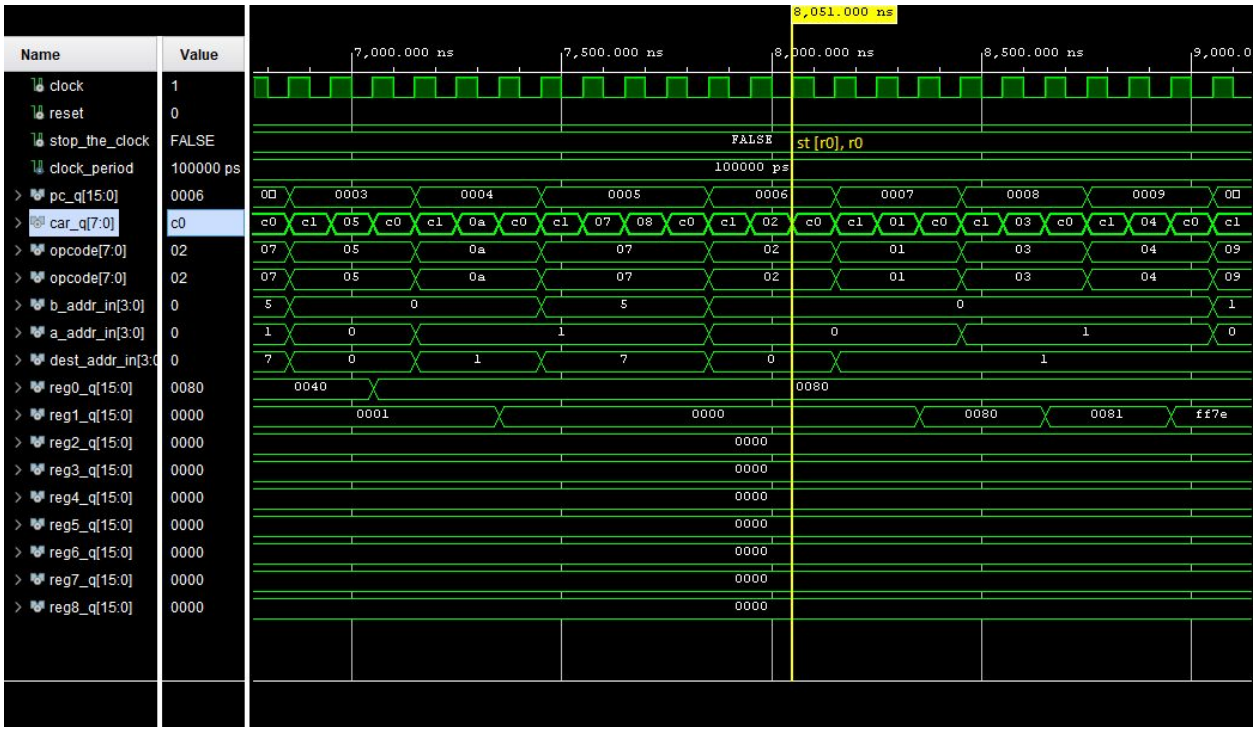
The above image illustrates the **bne** or branch not equal (conditional jump) instruction implemented. It checked if register 1 is not 0. Since it is not 0, it branched back to the two instructions to instruction (**add** `r0, r0, r0`).

DEC



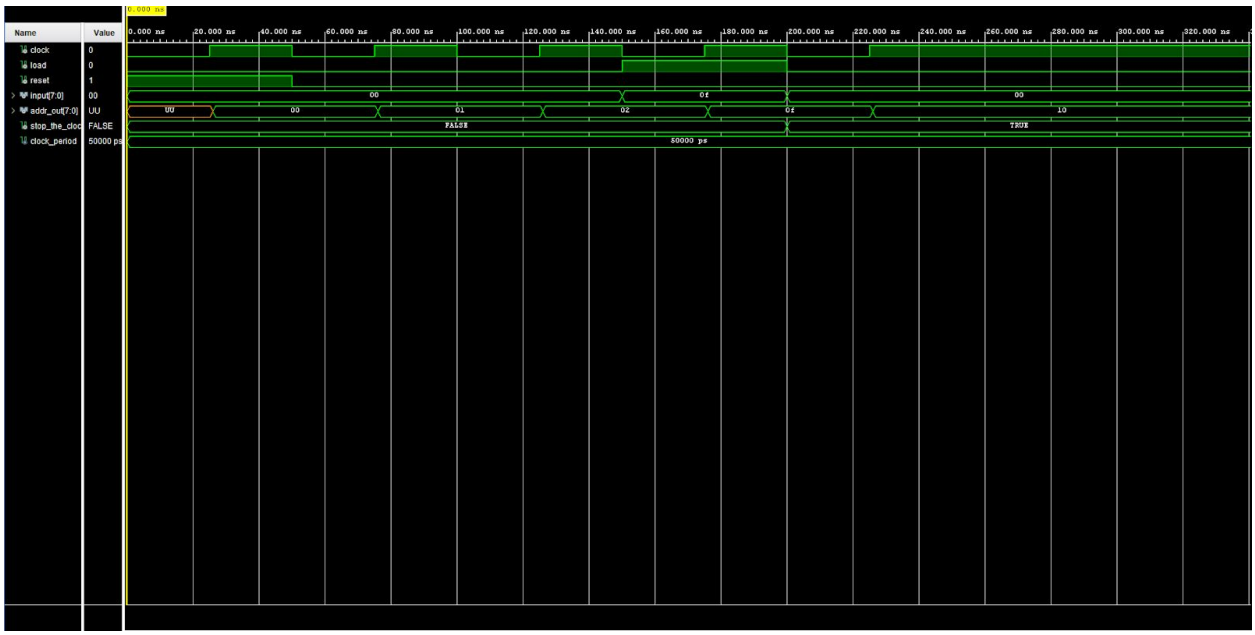
The above image illustrates the decrement (**dec**) instruction implemented. It decremented the number (0x7) in register 1 and stored the result (0x6) into register 1

ST



The above image illustrates the store (**`st`**) instruction implemented. It stored the number (0x0080) in register 0 into the memory location (0x0080) stored in register 0.

Control address register

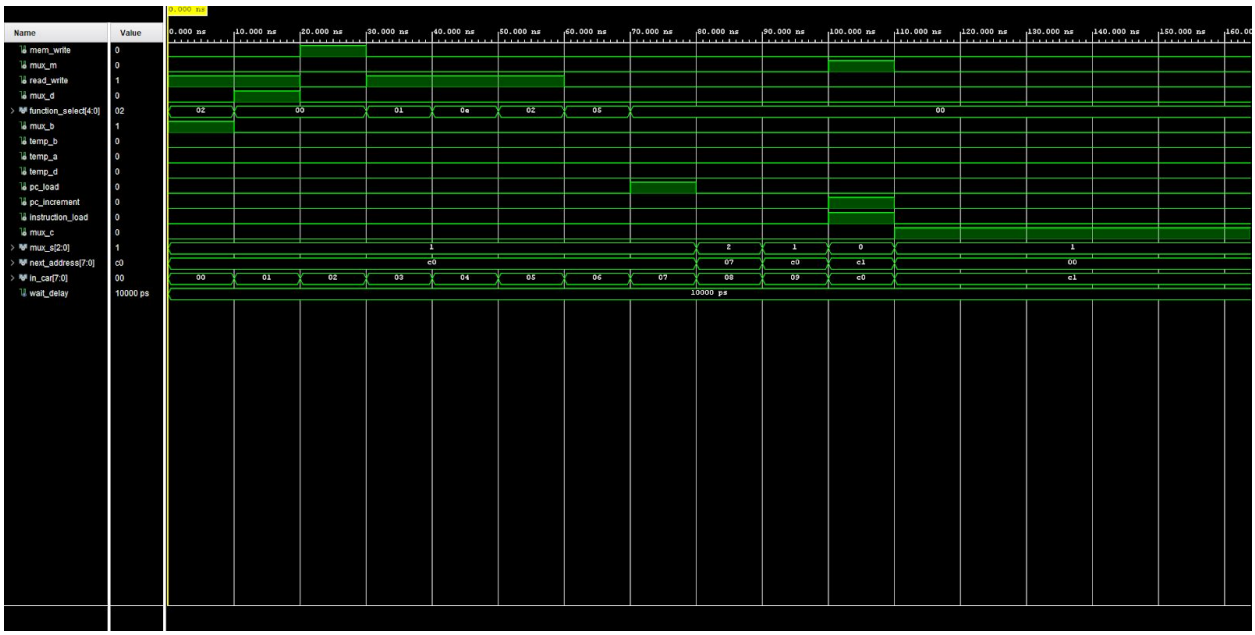


The control address register (CAR) has the ability to:

- Set the current address to 0x00 when reset is ‘high’.
- Increment the current address when load is ‘low’.
- Load the address from the input into the current address when load is ‘high’.

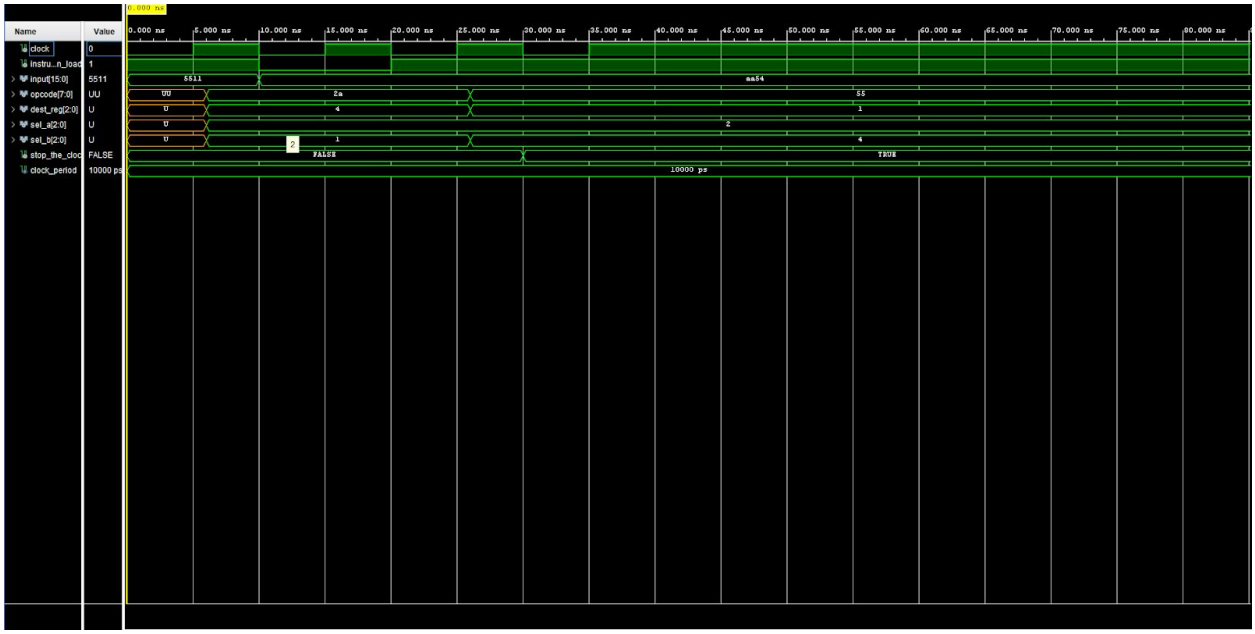
The test bench initially resets the CAR, increments twice, loads the input address and finally increments the current address.

Control memory



The control memory decodes the opcode (CAR output address) and outputs the decoded microcoded instruction. The testbench goes through all the microcoded instructions implemented. The instructions implemented, in order of appearance, are **`ADI, LD, ST, INC, NOT, ADD, B, BNE, NOP, SR, DEC, IF, EX0`.**

Instruction register

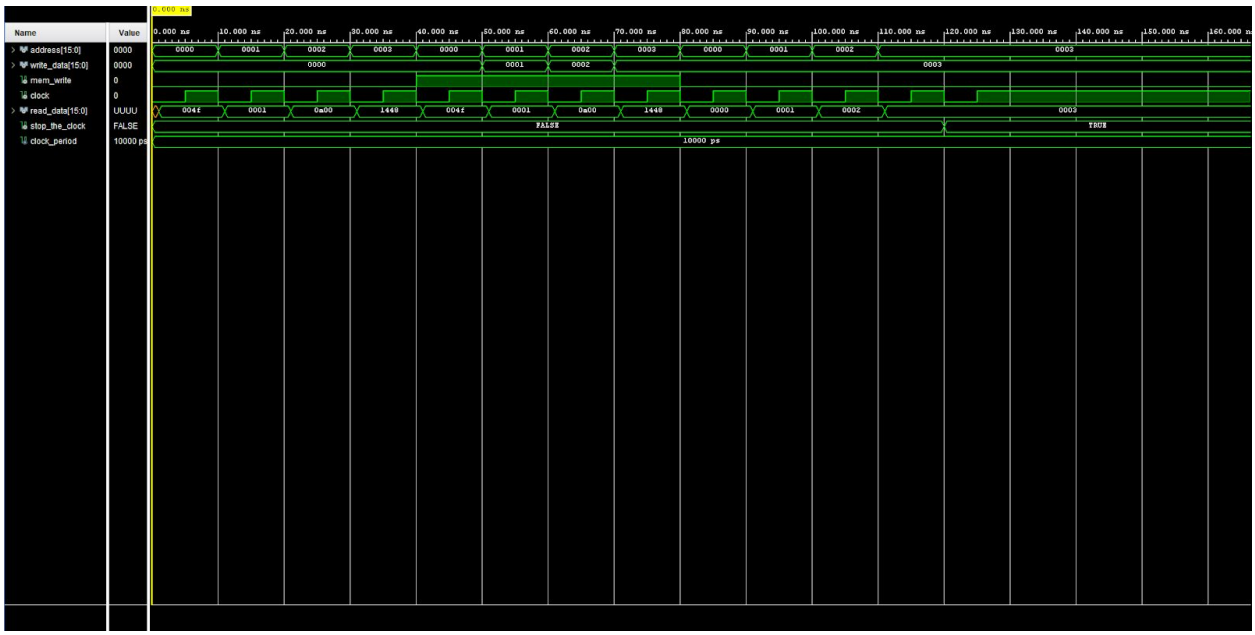


The instruction register simply splits up the input instruction and outputs the 8 bit opcode, destination register, source a register and source b register. The instruction register only changes when instruction load (IL) is ‘high’. The instruction register holds the instruction for n clock cycles until the microcoded instruction is completed.

The inputs of the instruction register test bench are as follows:

```
input <= b"0101010_100_010_001";
instruction_load <= '1';
wait for clock_period;
input <= b"1010101_001_010_100";
instruction_load <= '0';
wait for clock_period;
input <= b"1010101_001_010_100";
instruction_load <= '1';
wait for clock_period;
```

Memory



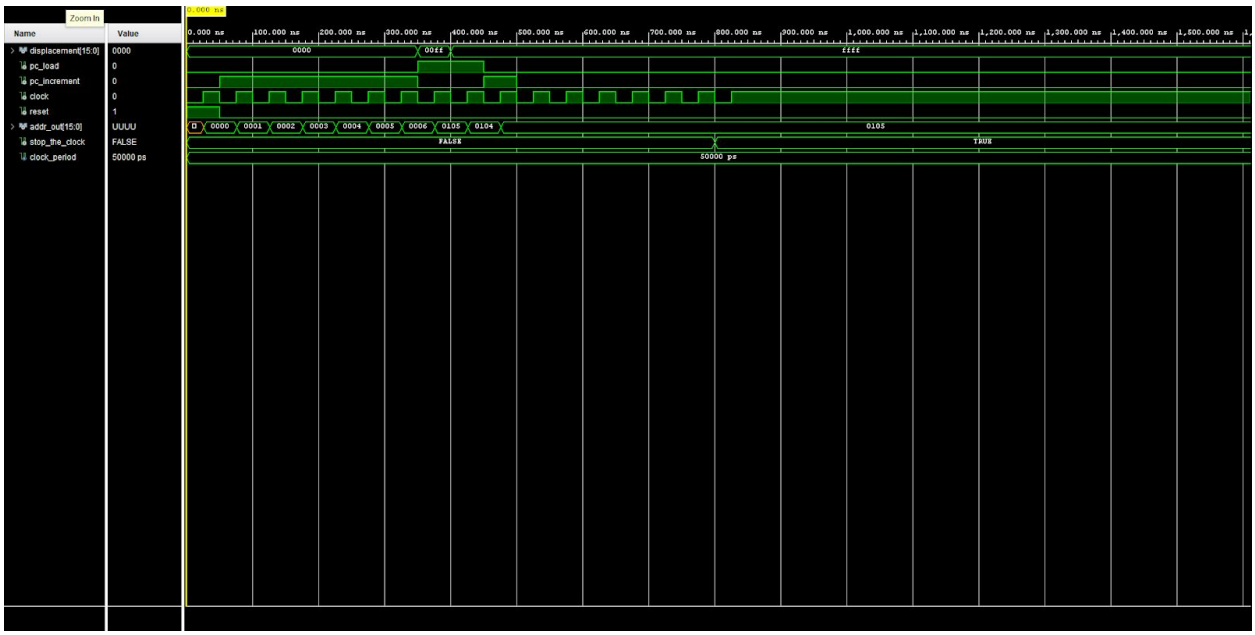
The memory contains the machine code that the microprogrammed instruction set processor will run. It can also contain arbitrary numbers which can be stored using the **`st`** instruction. Using the **`ld`** instruction allows the programmer to retrieve numbers within the memory.

The memory has the ability to:

- Output the number located at the input address.
- When memory write (MW) is ‘high’, the write data is written onto the memory at the location specified by the input address.

The test bench outputs the numbers at location 0x0 to 0x3, writes the numbers 0 to 3 into locations 0x0 to 0x3 and finally outputs the numbers at memory location 0x0 to 0x3 to verify if the memory write occurred.

Program counter

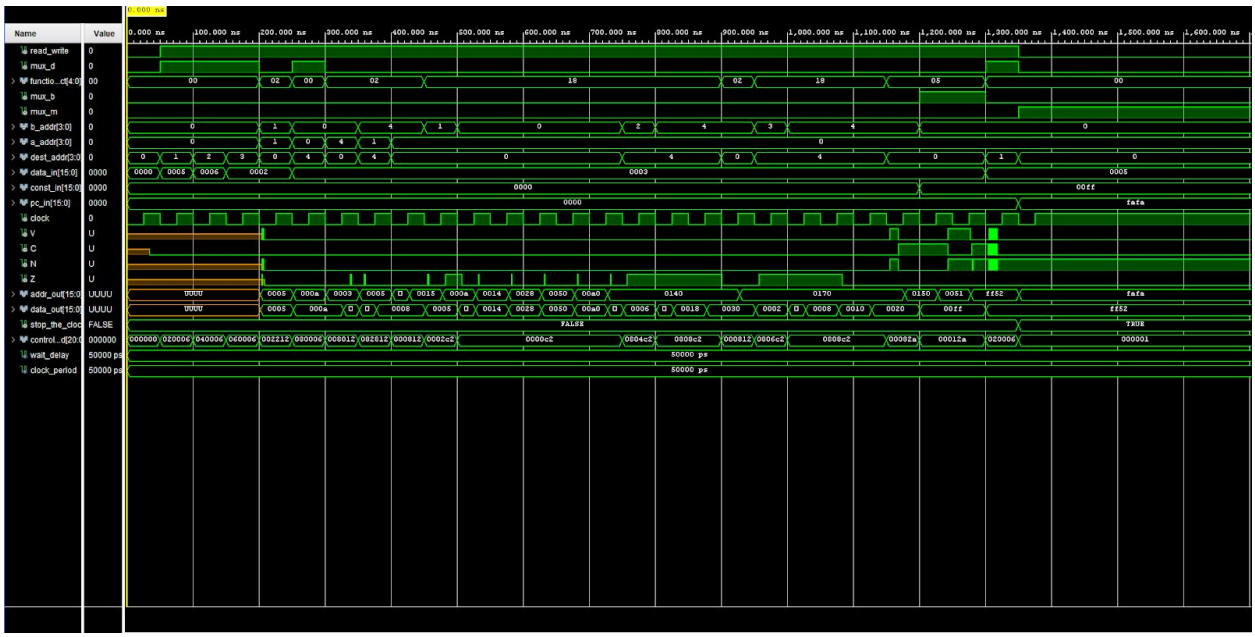


The program counter holds the next instruction in the memory to be executed.

The program counter has the ability to:

- Increment the program counter when program counter increment (PI) is ‘high’ and program counter load (PL) is ‘low’.
- Add the displacement to the program counter when program counter increment (PI) is ‘low’ and program counter load (PL) is ‘high’.
- Set the program counter to 0x0000 when reset is ‘high’.

Datapath



The datapath has been modified from the last assignment and now includes the updated register file which contains 9 registers. It also now contains a mux M and the pc_in. The inner components have been tested thoroughly in the previous assignment.

The datapath testbench implements the following arm assembly pseudo code:

```
mov r1, #5          ; x = 5
mov r2, #6          ; y = 6
mov r3, #2          ; z = 2
add r0, r1, r1      ; result = x + x
mov r4, #3          ; temp = 3
add r0, r4, r0      ; result += 3
add r4, r1, r4      ; temp += x
add r0, r0, r4      ; result += temp
; should be 21
; result = 64x + 8y - 16z
mov r0,r1, lsl #1   ; result = 2x
mov r0,r0, lsl #1   ; result = 4x
mov r0,r0, lsl #1   ; result = 8x
mov r0,r0, lsl #1   ; result = 16x
mov r0,r0, lsl #1   ; result = 32x
mov r0,r0, lsl #1   ; result = 64x
```

```
mov r4,r2, lsl #1  ; temp = 2y
mov r4,r4, lsl #1  ; temp = 4y
mov r4,r4, lsl #1  ; temp = 8y
add r0,r0,r4       ; result = 64x + 8y
mov r4,r3, lsl #1  ; temp = 2z
mov r4,r4, lsl #1  ; temp = 4z
mov r4,r4, lsl #1  ; temp = 8z
mov r4,r4, lsl #1  ; temp = 16z
sub r0,r0,r4       ; result = 64x + 8y - 16z
; should be 336
sub r0, #255       ; result -= 255
sub r0, #255       ; result -= 255
; result = 3x + 3 + 3
mov r1, #5         ; x = 5
; addr_out = pc_in
```

The final result should be $336_{10} - 255_{10} - 255_{10} = -174_{10} = 0xff52_{16}$.